Modeling and Applying the Knowledge of Synthesizer Patch Programmers

Pierre-Yves Rolland, François Pachet
Laforia-IBP, Université Paris 6, Boîte 169
4, place Jussieu,
75252 Paris Cedex 05, France,
Email: {rolland, pachet}@laforia.ibp.fr

Astract

In order to produce better man-machine interface for commercial synthetisers, we propose to automate substantial chunks of superficial knowledge related to patch programming. This knowledge is essentially related with how to *transform* sound patches rather than how to build patches from scratch. Sounds are therefore classified according to the transformations they can support. The classification is represented within the framework of *description logics*.

1. Introduction

Our framework stems from the following remark. In an intensive care unit, a typical nurse knows perfectly well how to use an infusion pump. However, the nurse may be an "expert" in infusion pump manipulation without necessarily having any theoretical knowledge of the device nor of medical issues related to diagnosis, treatment of patients, and so on. She may even be more expert than the doctor himself. But she knows what she needs to know: some kind of superficial knowledge about how to use the machine productively and safely. By analogy, we compare 1) expert *patch programmers*, who know how to effectively program synthesizers, and 2) *sound synthesis* experts, who possess extensive theoretical knowledge on, say signal processing and filtering but little knowledge on how to program a commercial synthesizer such as a Yamaha SY99, or Korg 05R/W. We focus on representing the knowledge manipulated by the first category of experts, and claim that it is possible to capture substantial chunks of this knowledge, to produce better man-machine patch programming interfaces.

Various CAS systems have been proposed, such as CHANT (Rodet et al. 1984), ISEE (Vertegaal and Bonis 1994), Kyma/Platypus (Scaletti 1989), Javelina (Hebel 1989), DMIX (Openheim 1989), or ARTIST (Miranda 1994). Most of these systems address sophisticated synthesis techniques, which are far less common in the community of real musicians than commercial synthesizers. Further, no system to date proposes to represent the common sense knowledge experts have on patch programming, particularly their knowledge on how to transform a patch into another. This is what motivates our attempt to study *CASP* (Computer-Aided Synthesizer Programming), as opposed to CAS in general. We will now detail the two assumptions on which our approach to CASP is based.

2. Assumptions

A1. Most of the expertise lies in the transformation knowledge.

It is a well known fact that synthesizer experts have trouble teaching how to make a sound from scratch. However, they are much more at ease in explaining how to transform one sound into another. Based on this observation, we think that most of the expertise lies in the transformation knowledge. This leads us to represent the knowledge associated to transformations, rather than to sound structures themselves. Following are two typical examples of transformation rules that we want our system to take into account. (R1) is a typical transformation rule that may apply to virtually any kind of commercial synthesizer providing parameters for a filter section. The second one, (R2), also applies to most commercial synthesizers, but is valid only for "harmonic" sounds, as opposed to "unpitched" tones such as cymbal tones:

(R1) You can make a sound b*righter* by increasing the low-pass filter's cutoff frequency.
(R2) You can make a sound *warmer* by duplicating the sound and applying a slight detune - typically 1/10 tone - to the duplicate.

A2. Commercial synthesizer programmers use superficial knowledge.

Our hypothesis is that commercial synthesizer programmers use far more know-how than theoretical knowledge. The case of FM synthesis is particularly interesting in this respect. The success story of the DX series showed that lots of famous patches were designed by people who understood only a limited fraction of the underlying - and complex - FM theory. Instead, they would use some kind of superficial knowledge about the complex interactions between the DX parameters, gained from experience or by studying other patches. Attempts to widespread FM theory by e.g. Chowning and Bristow (1986) showed that understanding FM theory played a minor role in programming effectively DX synthesizers. Although the book contains some theoretical material, the emphasis is put on giving practical hints to musicians. Programmers can use "know how" rules like R1 and R2 to build sounds, without knowing their theoretical meaning.

3. Transformation rules and sound classification

3.1. Organization of knowledge

Transformation rules such as the ones above are easy to represent in a knowledge-based system, e.g. by means of production rules. Of course, as the examples show, not all transformations are applicable to all kind of sounds. More precisely, transformations are binary i.e. each applies to one *origin sound types* and leads to one *target sound type*. Target sound types are characterized here by simple comparatives, such as "warmer", "brighter", etc. In order to organize those sound types, we need some kind of classification scheme. Classifying target sounds, based for instance on results of work in the area of timbre perception (Wessel 1979; Grey 1975), is irrelevant in our context: we are interested in classifying sounds according to what future transformations they can afford, and not according to the transformations that produced them. For example, we are not interested in classifying "brassy" sounds within a given typology of sounds. Rather, we are interested in identifying "sounds-that-may-become-brassy", i.e. sounds for which we know of a particular transformation that can make them "brassy".

In this scheme being an instance of a sound type means nothing more than "being able to undergo the transformations associated to that particular sound type", so sound names are not significant. However, since we need names (e.g. for browsing purposes), by convention we name sound types by suffixing them with "able", e.g. "brassy-able" or "warm-able". When a sound type affords several transformations, we give it a compound name, such as "brassy-and-bright-able". In this classification each transformation is associated to a particular sound type: transformation (R2) is associated to sound type "warm-able", (R1) is associated to "brassy-bright-able", and so forth.

The hierarchical relation linking a sound type to one or more parent sound types is a specialization relation. As a consequence, transformations associated to a sound type are inherited by its children. A part of a sound hierarchy for the Korg 05R/W synthesizer is shown on Figure 1. Sound types are represented as boxes, while names under boxes indicate applicable transformations. Arrows represent type/sub type relationships.

The main task of the system is therefore to classify a sound according to a pre-defined classification. We will now describe our framework, in which we represent sound types.

3.2. The representation framework

The characteristics of the knowledge we want to represent about sound transformations led us to look for a representation framework integrating both 1) classification facilities to manipulate sound hierarchies and 2) a procedural language to express transformations of the current sound.

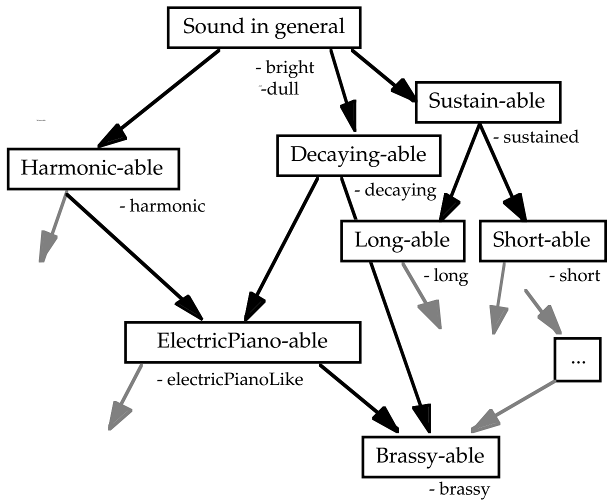

Figure 1. A part of the sound type hierarchy for the 05R/W synthesizer

Description logics to classify sounds.

An important characteristics of the sounds we want to model is that — at least on a superficial level — they may be described in a symbolic fashion, and not simply as an array of parameter values. For instance, a sound in the Korg 05R/W decomposes into one to sixteen *voices*. Each voice in turn decomposes into one or two *tones*, each characterized by a *filter section* , an *amplification section*, and so forth (KORG).

This strong structuring of sounds is well captured by *description logic (DL)* formalisms in which structural relations are stated in a declarative manner by the user and the system handles all inferences, i.e. classification and subsumption. A hierarchical knowledge organization is proposed in which every *concept* inherits information from more general ones. Also, the basis for a structuring of concepts is provided under the form of inter-concept links or *roles*. Among the various available DL implementations (Heinson et al. 1994), we were particularly interested in the BACK system (Hoppe et al. 1993), which offers a satisfactory compromise between expressiveness, completeness and efficiency. In Back, concepts are sets of objects specified either intensionaly or extensionaly, and objects are instances of one or more concepts. Roles are binary relations between objects. For any given object o, the objects that are linked to o through role r are called *role fillers* for role r and object o (see examples below). Providing information to BACK V5 can be done through term introductions and object creations, or through the use of *non-definitional rules* which impose particular logical relations between given concepts and roles. Besides, various kinds of information can be retrieved: retrieving the result of the classification for an object *o* yields the list of concepts *o* instantiates; querying the system for concept subsumption produces a Boolean answer as to whether a concept c1 is *subsumed* by concept c2, i.e. whether any instance of c1 is also an instance of c2.

OOP to represent transformations.
On the other hand, we need to represent transformations effectively and not only at an abstract level. As will be seen, the formalism of Description logics does not allow to represent and organize transformations easily. Moreover, MIDI communication as well as user interface are typically a lot easier to program using object-oriented techniques. Sounds can be represented as instances of sound classes, and transformations as methods for these classes. We chose Smalltalk-80 for its acknowledged ease of use, and for its widespread use in the computer music research community (see Computer Music Journal, 13 (2), 1989 for instance).

A scheme to link both representations.
Having two different knowledge representation paradigms coexist is not straightforward because information is redundant and incompatible: A sound is represented both as a BACK object and as a Smalltalk instance, which are not directly compatible. We propose a scheme for coupling the two representations (section 5).

4. The classification oriented representation of sounds

We built up a representation of sounds using the formalism of Description Logics, by introducing two kinds of concepts: *fundamental* and *abstract* concepts. fundamental concepts are used to represent the various entities shown on Figure 2; abstract concepts represent the structural part of the programmer's expert knowledge. We will now examine these two concept categories in more detail.

Fundamental concepts
We represent the technical description of sounds (as provided by the synthesizer manufacturer) by a set of Back terms. We call these terms "fundamental" because they are a quasi direct transcription of the synthesis model architecture of the synthesizer.
For example, the `tone` concept cannot be defined using less specific concepts other than a special preset concept called `anything`. Therefore, the `tone` concept is introduced using necessary classification conditions with respect to the classification of its instances, which is materialized by a primitive concept introduction, denoted by the symbol :< as follows:

```
tone :< anything
```

The `waveform` concept is *defined* (symbol :=) as its extensive introduction yields necessary and sufficient instance classification conditions. For obvious reasons, instead of listing all possible waveforms (over 300), we provide here an excerpt with just a few examples.

```
waveform := attribute_domain ([sine, square,
saw, organ, fluteLoop, whiteNoise]).
```

The primitive role `hasWaveform` can then be introduced to represent which waveform a tone is based on. We call such a role *terminal* as its fillers represent actual synthesis parameters as opposed to abstract structures like voices or envelopes.

```
hasWaveform :< domain(tone) and
range(waveform).
```

Another defined concept is `doubleVoice`, which is introduced based on the primitive concept `voice`, with *number restrictions* applying to the role `hasTone`:

```
doubleVoice := voice and exactly(2, hasTone).
```

Abstract concepts
On top of this first layer of representation, we build up a hierarchy of concepts that represent the structural part of the programmer's expertise. As will be explained later, the concepts introduced here are used to define transformations. We divide these concepts into three categories:

Abstract concepts built from fundamental concepts.
These include partial descriptions of sounds, such as `heldTimeFunction`, defined as a `TimeFunction` whose 'sustainLevel' value is greater or equal to 1. In synthesis terms, this allows to describe, for instance, sounds whose loudness eventually stabilizes to a non-zero value. On the 05R/W, this is obtained by setting the Time Variant Amplifier Envelope Generator's Sustain Level to a strictly positive value.

```
timeFunction  and the (sustainLevel, ge(1))
    => heldTimeFunction.
```

Similarly, we introduce abstract concepts that, as will be seen below, play a part in building up the brassy-able sound type's representation. This code listing shows the manner in which these interdependent, abstract concepts are introduced.

```
FilterEnveloppe      and heldTimeFunction
    and the(attackTime, ge(17) and le(25))
    and the(attackLevel, ge(85))
    and the(decayTime, ge(60) and le(75))
    and the(intermediateLevel, ge(25) and
    le(35))
    and the(heldLevel, ge(25) and le(35))
    => brassyAbleFilterEnv.

filter and
  the(hasEnveloppe,brassyAbleFilterEnv)
    => brassyAbleFilter.

brightTone
    and the(hasFilter, brassyAbleFilter)
    and the(hasAmp, brassyAbleAmp)
    => brassyAbleTone.
```

```
voice  and atleast (1, hasTone, brassyAbleTone)
    => brassyAbleVoice.
```

Other examples of abstract concepts which reflect structural expert knowledge are those describing 'non transformable' sounds. Contrary to the above abstract concepts, these concepts provide *complete* descriptions of sounds which do not afford any particular transformation but which are used for describing transformable sounds:

```
sound  and no(hasVoice,
    voiceWithInharmonicTone)
    => harmonicSound
```

Note that since any sound type is subsumed by the transformable sound type `soundInGeneral`, even instances of non transformable sounds types can undergo some general transformations, such as 'make bright' or 'make dull'**.**

Concepts describing transformable sounds.
Finally, these concepts describe the sounds types found in the hierarchy (figure 1). Here are a few examples of such concepts.

```
harmonicSound and some (hasVoice,
brassyAbleVoice)
    => brassyAbleSound
sound  and all(hasVoice, sustainAbleVoice)
    => sustainAbleVoice.
sound  => soundInGeneral.
```

Concepts representing transformations themselves.
To each sound type we associate a list of transformations, represented as mere character strings. This list is materialized by a sub-concept of `possible-Transformations`, an extensional concept which lists all possible transformations for all existing sound types. Here are a few examples:

```
possibleTransformations:= attribute_domain
([warm, brassy, dull, decaying (...)]).

affordsTransformation :< domain(sound) and
range (possibleTransformations).

brassyAbleSound :< affordsTransformation :
brassy.

WarmAndSharpAbleSound :< affordsTransformation:
warm and sharp.
```

5. Integration scheme

Representing sounds as objects, in the sense of object-oriented programming, is particularly natural in our context, where emphasis is put on transformations. Each transformation is represented by a Smalltalk method defined in class `CurrentSound`.

The integration scheme needed to link these two representations is based on two principles: fundamental concepts on one hand are represented by Smalltalk classes. Each role of a concept is represented by an instance variable of the corresponding class. An actual sound is represented by an instance of class `CurrentSound`, and its parameters by instances of the corresponding classes. Transformations, on the other hand, are represented by methods associated to class `CurrentSound`. Each method modifies the current sound by changing some values of its parameters. Therefore, the semantics of the BACK symbol representing transformations is given by the corresponding Smalltalk method. This yields a two-level representation framework with two links, as shown in Figure 2.

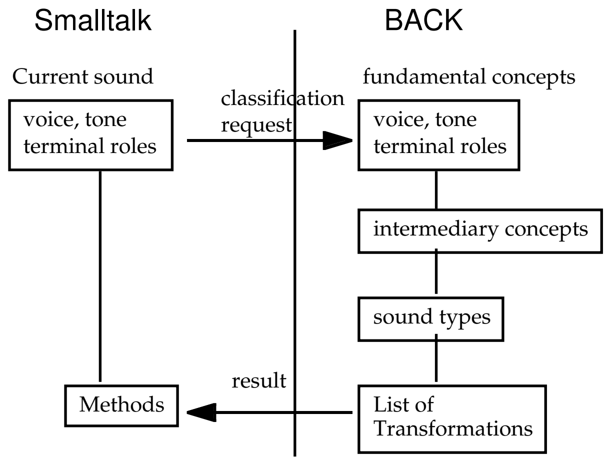

Figure 2. The two representations of sounds and their connection.

6. Execution

A session with our system is described by the following iterative cycle of operations :
Step 1. The user chooses an initial patch. This initial patch may be either one of the user's own patches or one selected from an external library, including patches created during past sessions.
Step 2. The patch is transmitted to BACK to be classified. As a result, BACK generates the list of sound types instantiated, together with the associated transformations. This data is then transmitted to Smalltalk.
Step 3. The user selects one of the proposed transformations. Parameters in the `currentSound` Smalltalk object are modified accordingly, and actual synthesizer parameters are changed so the user can play and listen to the new sound.
Step 4. Back to step 2, and loop until the user is satisfied with the current sound.

Figure 3 shows an example of a typical user session. In order to provide some flexibility, our system offers an alternative to step 3 in which the user directly changes individual synthesizer parameters. This still allows for the resulting sound to get classified and thus undergo further transformation cycles.

7. Discussion

The main contribution of this work concerns the use of a sophisticated AI representation formalism, Description Logics, to capture superficial knowledge about synthesizer patch programming. This knowledge, mainly related to sound transformations, can then be exploited to help musicians browse through the timbre space of a commercial synthesizer in an intuitive way, thereby reducing the complexity of the sound making process.

A prototype was built and tested with the Korg 05R/W synthesizer. In a typical session the user iteratively selects transformations proposed by the classification process. After a transformation is applied, the current sound gets re-classified, which makes it able to undergo new transformations, and so on. The work is still in progress, and our efforts concentrate on providing the system with a learning capability (e.g. based on inductive learning from examples), which will allow the user to define new sound types (resp. transformations) by presenting sets of example patches (resp. patch couples) to the system.

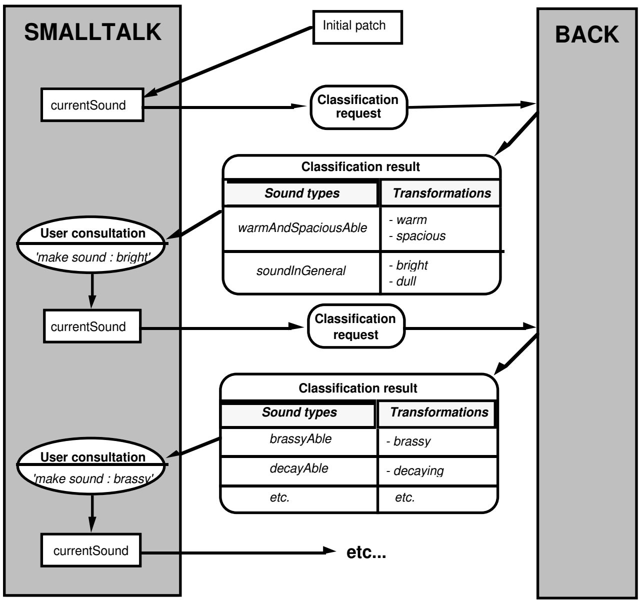

Figure 3. An example session.

8. References

Brachman, R.J., and J.G. Schmolze, 1985. "An overview of the KL-ONE knowledge representation system." *Cognitive Science* 9(2): 171-216.

Chowning, J. and D. Bristow. 1986. *FM Theory & applications by musicians for musicians*. Yamaha Music Foundation Corp.

Goldberg, A., and D. Robson. 1983. "Smalltalk-80 : The language and its implementation." Reading, MA: Addison-Wesley.

Grey, J. 1975. *An Exploration of Musical Timbre*. Ph.D. dissertation, Stanford University Psychology Dept. CCRMA Report STAN-M-2.

Hebel, K, 1989. "Javelina: An Environment for Digital Signal Processing Software Development." *Computer Music Journal* 13(2) Summer 1989. Reprinted in S. Pope (Ed.) *The Well-Tempered Object*, Cambridge, MA: MIT Press, 1991.

Heinsohn, J. Kudenko, D. Nebel, B. Profitlich, H.-J. 1994. "An empirical analysis of terminological representation systems." *Artificial Intelligence* 2: 367-397. Germany: Elsevier.

Hoppe, T. C, Kindermann, J. Quantz, A. Schmiedel, M. Fischer. 1993. *Back V5 Tutorial & Manual*, Institut fûr Software und theoretische Informatik, W-1000 Berlin 10, Germany, march 1993.

KORG. Undated. Korg 05R/W *Manuel d'utilisation*, *AI*^{2} *Synthesis Module*. Korg Inc.

Michalski, R. S. 1983. "A theory and methodology of inductive learning." in Michalski, R.S., J.G. Carbonell and T.M. Mitchell, (eds.), *Machine Learning: an Artificial Intelligence approach*. Palo Alto, California: TIOGA Publ. Co. pp. 83-134.

Miranda, E. 1992. *From symbols to sounds: an AI-based investigation of Sound Synthesis (Ph.D. Thesis Proposal).* DAI Discussion Paper, No. 117, Dept. of Artificial Intelligence, University of Edinburgh.

Openheim, D.V. 1989. "DMIX: An Environment for Composition." 1989 *International Computer Music Conference*. San Francisco: Computer Music Association.

Openheim, D. V. 1991. "Shadow: An Object-Oriented Performance System for the DMIX Environment." Proceedings of the 1991 *International Computer Music Conference*, Montréal. San Francisco: Computer Music Association, pp. 281-284.

Risset, J.C. 1969. "An introductory catalogue of computer-synthesized sounds". Bell Telephone Labs.

Rodet, X., Potard, Y., Barrière, J.B. 1984. "The CHANT project : from the synthesis of the singing voice to synthesis in general." In C. Roads (ed.), *The Music Machine*, Cambridge, MA: MIT Press.

Scaletti, C. 1989. "The Kyma/Platypus Computer Music Workstation." *Computer Music Journal* 13(2): 23-38.

Serra, X. 1989. *A system for sound analysis/transformation/synthesis based on a deterministic plus stochastic decomposition.* Ph.D. Thesis, Stanford University.

Vertegaal, R., and E. Bonis. 1994. "ISEE : an intuitive sound editing environment." *Computer Music Journal* 18(2) 21-29.

Wessel, D. 1979. "Timbre Space as a Musical Control Structure." *Computer Music Journal* 3(2): 45-52.